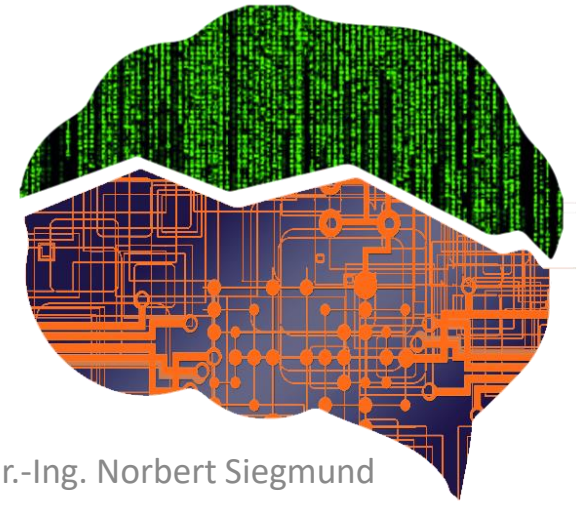
Software Engineering

Qualität, Wartung und Evolution

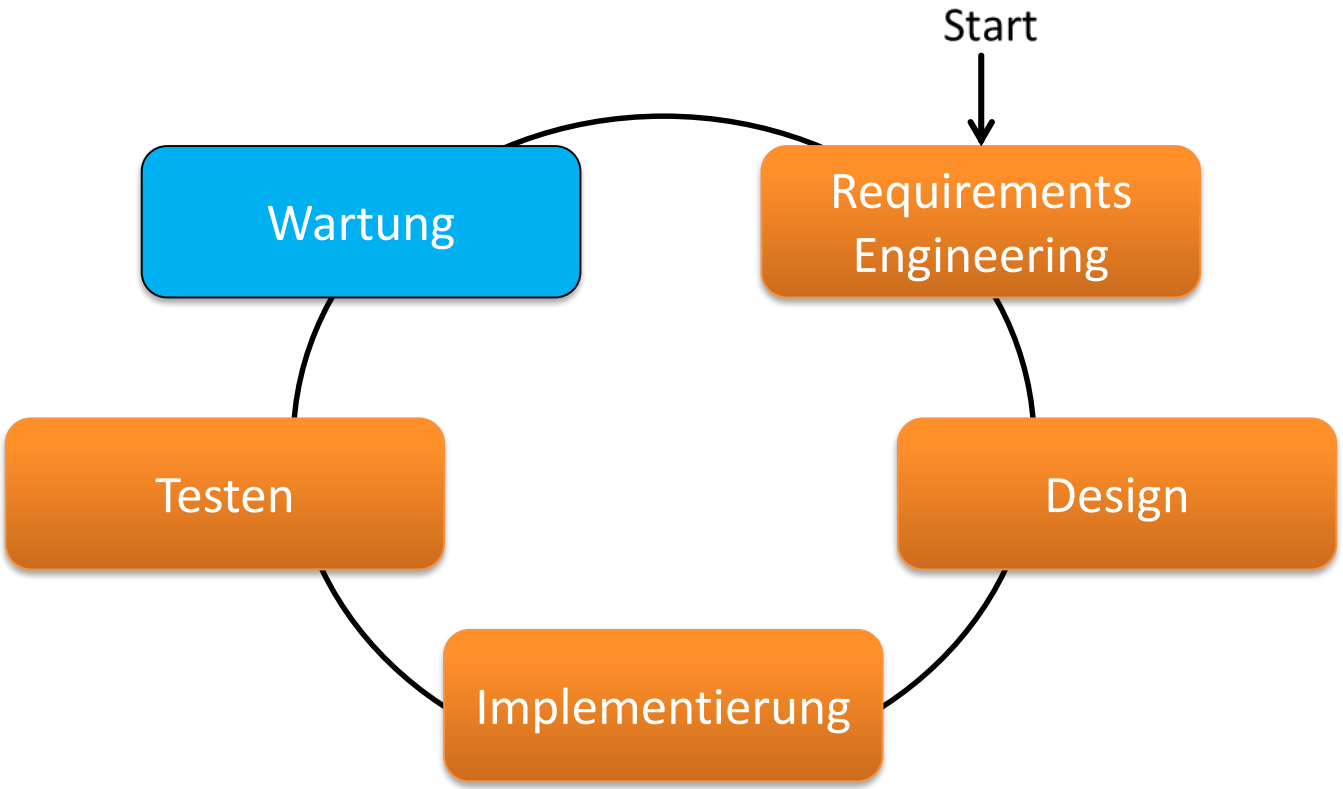


Prof. Dr.-Ing. Norbert Siegmund
Intelligent Software Systems

Bauhaus-Universität Weimar

Basierend auf dem Material von Oscar Nierstrasz, Sven Apel, Janet Siegmund

Einordnung


Start
Requirements Engineering
Design
Implementierung
Testen
Wartung

Lernziele

- Qualität von Software bewerten können
- Qualität auf Quelltextebene verbessern können
- Rolle von Wartung im Softwarelebenszyklus verstehen

- 2. Teil der Vorlesung: Software Evolution

Software Qualität

Aufgabe

- Was ist Software Qualität?
- Wie würden Sie Software Qualität messen?

Software Qualität

- Korrekte Implementierung der Anforderungen
- Design-Qualität:
  - Erweiterbarkeit, Wartbarkeit, Verständlichkeit, Lesbarkeit, Wiederverwendbarkeit,…
  - Robustheit gegen Änderungen
  - Modularität, Low Coupling, High Cohesion
- Zuverlässigkeit, geringe Fehleranfälligkeit, Testbarkeit, Performance
- Usability, Effizienz, Erlernbarkeit

Software Qualität: Probleme

- Spannung zwischen Stakeholdern
  - Kunde: Effizienz, Usability
  - Entwickler: Wartbarkeit, Wiederverwendbarkeit, Verständlichkeit
- Wie misst man eigentlich Software Qualität?
  - „You can't control what you can't measure"*
  - Software-Metriken? -> direkt messbar
  - Empirische Untersuchungen? -> indirekt messbar

*Tom DeMarco, 1986

Software Metriken

- Lines of Code (LoC)
- Bugs per line of code
- Comment density
- Cyclomatic complexity (measures the number of linearly independent paths through a program's source code)
- Halstead complexity measures (derive software complexity from numbers of (distinct) operands and operators)
- Program execution time
- Test Coverage
- Number of classes and interfaces
- Abstractness = ratio of abstract classes to total number of classes
- …

Software Metriken werden selten benutzt

- Einige Firmen erstellen Messprogramme, noch weniger haben Erfolg damit
- Firmen, die Metriken benutzen, machen das oft nur, um bestimmten Qualitätsstandards zu genügen
  - Bei Interesse: N. E. Fenton, "Software Metrics: Successes, Failures & New Directions”, 1999.

- Hier kommt wieder der Unterschied zur Ingenieursdisziplin zum Tragen
  - Ingenieure sind erfolgreich, weil sie Qualität gut messen können
  - Geht eben nicht bei Softwareprodukten

Software Metriken sind selten sinnvoll

- Formal definierte Metriken sind zwar objektiv, aber was bedeuten sie?
  - Was sagt eine zyklomatische Komplexität von 12 über die Qualität des Quelltextes aus?
  - Gar nichts

- Oft unklar, ob Metrik mit irgendeinem sinnvollen Qualitätskriterium zusammenhängt
  - Wäre so, als würde man Intelligenz mit Hirnmasse oder Kopfumfang messen wollen

Software Qualität

- Wenn Metriken nicht sinnvoll sind, wie dann Software-Qualität messen?
- Analyse für jedes Softwareprojekt
  - Design-Qualität, Erweiterbarkeit
  - Vergleich zu anderen Designs, die sich als sinnvoll erwiesen haben (z.B. Design Patterns)
  - Code smells und Antipatterns suchen
  - Umfangreiche Tests und Code Reviews
  - (Formale) Verifikation
  - Nutzerstudien

Qualitätsmerkmale

- Beziehen sich auf Produkt und Prozess
  - Produkt: an Kunden geliefert, externe Qualität
  - Prozess: produziert das Produkt, interne Qualität

- Qualitätsprozess ist notwendige Bedingung für ein Qualitätsprodukt

Qualitätsmerkmale

- Korrektheit (siehe Testen-Vorlesung)
- Zuverlässigkeit (Wahrscheinlichkeit, dass System erwartungsgemäß läuft)
- Robustheit („gutes“ Verhalten in unbekannten Systemzuständen)
- Effizienz (geringer Ressourcenverbrauch)
- Usability (Maß der Bedienbarkeit, Nutzerkontrolle/-freiheit, Ähnlichkeit zur Welt)
- Wartbarkeit (wie einfach es ist das System zu ändern / anzupassen nach Release)
- Verifizierbarkeit (Einfachheit der Überprüfung gewünschter Eigenschaften)
- Verständlichkeit (Einfachheit des Verstehens für interne und externe Nutzer)
- Produktivität (wie produktiv sind Entwickler für Teile des Systems?)
- Pünktlichkeit (pünktliche Auslieferung)
- Transparenz (Einsehbarkeit des Projektzustands)

Aufgabe: NoMoreWaiting

- Bewerten Sie, wie wichtig die Kriterien für NMW sind
- Wie könnte man diese Kriterien messen?
  - Korrektheit, Zuverlässigkeit, Robustheit, Effizienz, Usability
  - Wartbarkeit, Verifizierbarkeit, Verständlichkeit, Produktivität, Pünktlichkeit, Transparenz

Qualität auf Implementierungsebene: Code Smells

- Jedes Symptom im Quelltext, das auf größeres Problem (z.B. Antipattern) hinweist
- Nicht jeder Code Smell ist schlechter Code; bei jedem Code Smell überprüfen, ob er wirklich behoben werden muss
- Typische Code Smells:
  - Duplizierte Code
  - Lange Methode/Klasse
  - High coupling/low cohesion
  - Schrotkugeln (shotgun surgery)

Qualität auf Implementierungsebene: Antipatterns

- Siehe Vorlesung Design Patterns
- Beispiele:
  - The Blob (Gottklasse)
  - Action at a distance: Unerwartete Interaktion zwischen eigentlich getrennten Modulen
  - Reihenfolge: Wenn Methoden in einer Klasse in bestimmter Reihenfolge aufgerufen werden müssen
  - Zirkuläre Abhängigkeit: Unnötige Abhängigkeiten zwischen Modulen
- Mehr Beispiele: http://c2.com/cgi/wiki?AntiPatternsCatalog

Systemqualität, Standards und Kontrolle

Produkt- und Prozessstandards

- Produktstandards spezifizieren *Eigenschaften, die alle Komponenten aufweisen müssen*
- Prozessstandards spezifizieren, *wie der Entwicklungsprozess ablaufen soll*

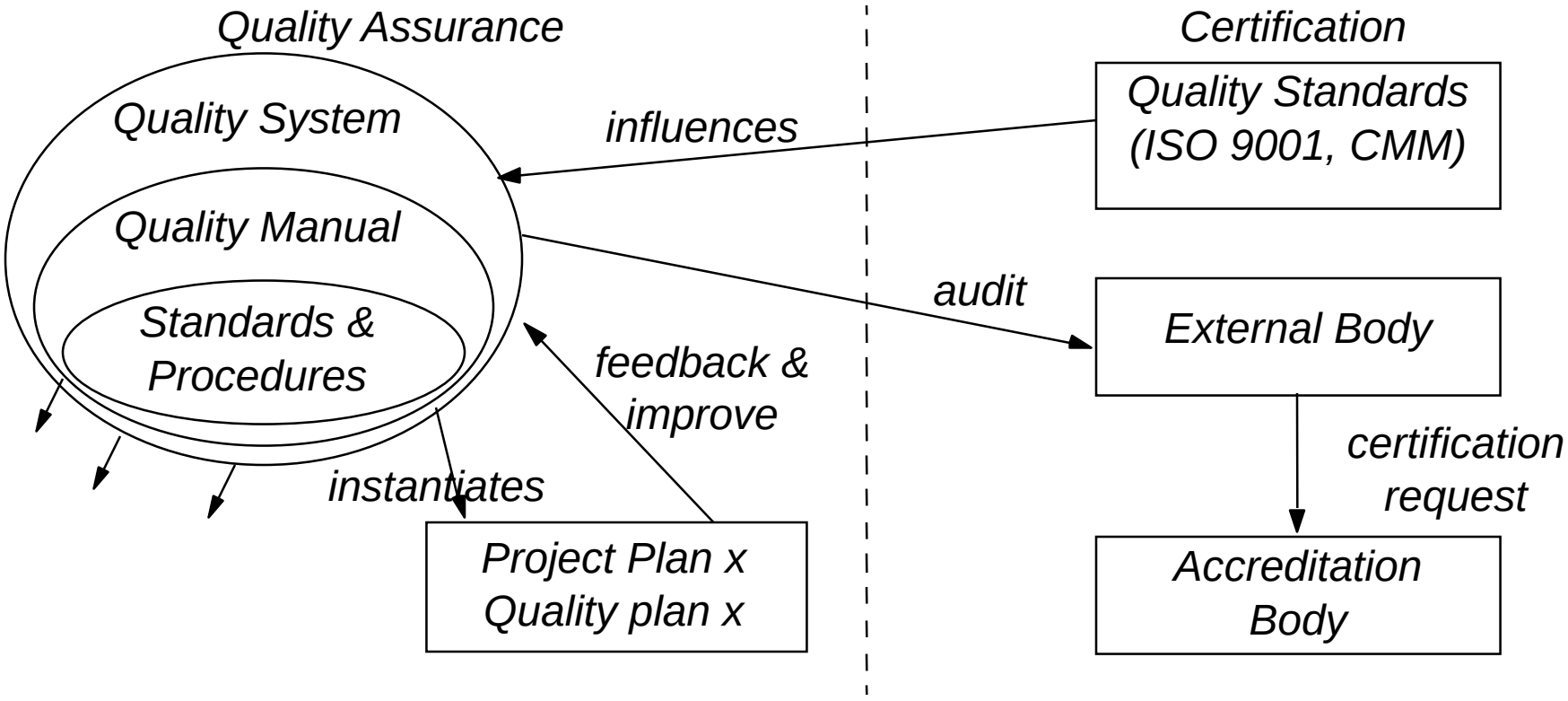
Product standards
Design review form
Document naming standards
Procedure header format
Java conventions
Project plan format
Change request form

Process standards
Design review conduct
Submission of documents
Version release process
Project plan approval process
Change control process
Test recording process

Potentielle Probleme mit Standards

- Wird nicht immer als *relevant und up-to-date* gesehen von Entwicklern
- Erfordert evtl. zu viel *Bürokratie* (ausfüllen von Formularen)
- Erfordert evtl. *aufwändige manuelle Arbeit*, wenn keine Tool-Unterstützung vorhanden ist
  - *Overhead limitiert effektive Anwendbarkeit von Standards*

Qualitätssystem



Kunden benötigen evtl. ein extern überprüftes Qualitätssystem

Qualitätsplan

Ein Qualitätsplan sollte:

- Die gewünschten Produkteigenschaften festlegen und wie diese gemessen werden können
- Die Qualität des Überprüfungsprozesses definieren
  - D.h., die Kontrollen, die die Qualität sicherstellen
- Die Standards festlegen, welche angewendet werden

- Qualitätsmanagement sollte vom Projektmanagement separiert werden, um Unabhängigkeit zu gewährleisten

Qualitätskontrolle

1. Reviews
   - *Inspektionen* für Fehlerbehebung
   - *Fortschrittserhebung*
   - *Qualitätsreviews*

2. Automatisierte Kontrollen
   - *Messen* Softwareattribute und vergleichen sie gegen Standards (z.B. Fehlerrate, Cohesion, etc.)

Typen von Qualitätsreviews

- Ein Qualitätsreview wird durch eine *Gruppe* von Menschen durchgeführt, die *jeden Teil des Softwaresystems* und deren *Dokumentation* begutachten

- Reviews sollten *aufgezeichnet* und *verwaltet* werden
  - Somit kann Fortschritt überprüft werden

Typen von Qualitätsreviews …

Review-Typ	***Zweck***
Formale, technische Reviews (bzw. Design- oder Programminspektionen)	Getrieben durch *Checkliste* • Erkennt detailiert Fehler in jedwedem Produkt • Erkennt Diskrepanz zw. Anforderung und Produkt • Prüft, ob Standards eingehalten werden.
Progress reviews	Getrieben durch *Budgets, Pläne und zeitl. Anforderungen* • Prüft, ob Projekt nach Plan läuft • Erforder präzise Meilensteine • Ist Prozess- und Produktreview

Review Guidelines

- Review das *Produkt*, nicht den Produzenten (Entwickler)
- Lege eine *Agenda* fest und halte dich daran
- *Begrenze Debatten* und Widerlegungen
- *Identifiziere Problemgebiete*, aber versuche nicht jedwedes Problem zu lösen
- Verwende *geschriebene Notizen*
- *Begrenze die Anzahl von Teilnehmern* und verlange eine Vorbereitung
- Entwickle eine *Checkliste* für jedes Produkt, welches begutachtet wird
- *Weise Ressourcen* und Zeit für Reviews zu
- Veranstalte sinnvolles *Training* für alle Reviewer
- *Begutachte* deine eigenen ersten Reviews

Sample Review Checklists (I)

Software Project Planning

1. Is software scope unambiguously defined and bounded?
2. Are resources adequate for scope?
3. Have risks in all important categories been defined?
4. Are tasks properly defined and sequenced?
5. Is the basis for cost estimation reasonable?
6. Have historical productivity and quality data been used?
7. Is the schedule consistent?

...

Wird nicht in Vorlesung besprochen, sondern dient der Dokumentation und Vorbereitung

Sample Review Checklists (II)

Requirements Analysis

1. Is information domain analysis complete, consistent and accurate?
2. Does the data model properly reflect data objects, attributes and relationships?
3. Are all requirements traceable to system level?
4. Has prototyping been conducted for the user/customer?
5. Are requirements consistent with schedule, resources and budget?

...

Wird nicht in Vorlesung besprochen, sondern dient der Dokumentation und Vorbereitung

Sample Review Checklists (III)

Design

1. Has modularity been achieved?
2. Are interfaces defined for modules and external system elements?
3. Are the data structures consistent with the information domain?
4. Are the data structures consistent with the requirements?
5. Has maintainability been considered?

...

Wird nicht in Vorlesung besprochen, sondern dient der Dokumentation und Vorbereitung

Sample Review Checklists (IV)

Code

1. Does the code reflect the design documentation?
2. Has proper use of language conventions been made?
3. Have coding standards been observed?
4. Are there incorrect or ambiguous comments?

...

Wird nicht in Vorlesung besprochen, sondern dient der Dokumentation und Vorbereitung

Sample Review Checklists (V)

Testing

1. Have test resources and tools been identified and acquired?
2. Have both white and black box tests been specified?
3. Have all the independent logic paths been tested?
4. Have test cases been identified and listed with expected results?
5. Are timing and performance to be tested?

Wird nicht in Vorlesung besprochen, sondern dient der Dokumentation und Vorbereitung

Review Ergebnisse

- Kommentare, die während einer Begutachtung gemacht wurden, sind *vertraulich*.
- **Keine Aktion**
  - Keine Änderung an der Software oder Dokumentation erforderlich
- **Reparatur bevorzugt**
  - Designer oder Programmierer sollten identifizierten Fehler korrigieren
- **Überdenken des gesamten Designs**
  - Das identifizierte Problem betrifft andere Teile des Systems und des Designs

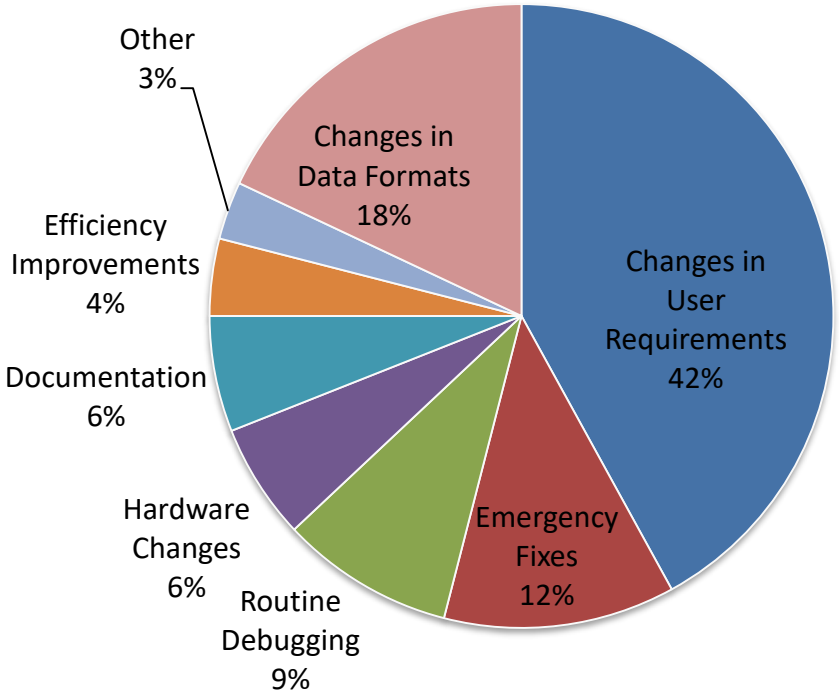
Wartung
-- Refactoring

Wartung

- Jede Änderung, die nach Release an der Software durchgeführt wird
- Korrigierend: Fehler beheben
- Perfektionierend: Anpassung an ändernde Anforderungen
- Adaptiv: Anpassung an neue Plattformen
- Präventiv: Vermeidung von potentiellen Problemen

Wartungskosten

- Bis zu 80% der Kosten für Softwareentwicklung
- Wartungsprogrammierer verbringen die meiste Zeit mit dem Verstehen von Quelltext (bis zu 60%)



http://swreflections.blogspot.de/2011/04/lientz-and-swanson-on-software.html

Antipatterns und Code Smells beheben

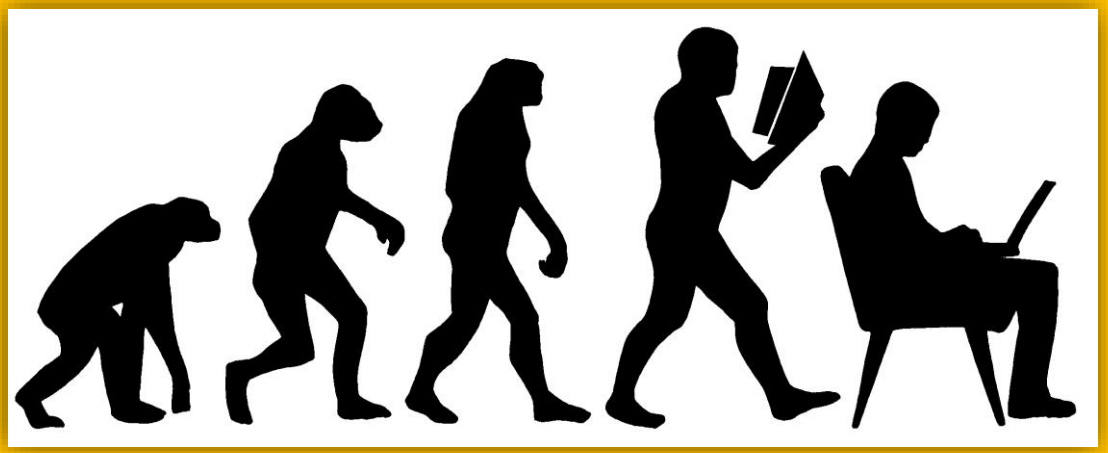
- Refactorings, um systematisch Codes Smells und Antipatterns zu beseitigen
- Duplizierter Quelltext (in zwei Methoden):
  - Private Hilfsmethode
  - Eclipse: Extract Method
- Duplizier Quelltext (in zwei Klassen):
  - In Elternklasse verschieben
  - Template Design Pattern für verschiedene Aufgaben

Antipatterns und Code Smells beheben

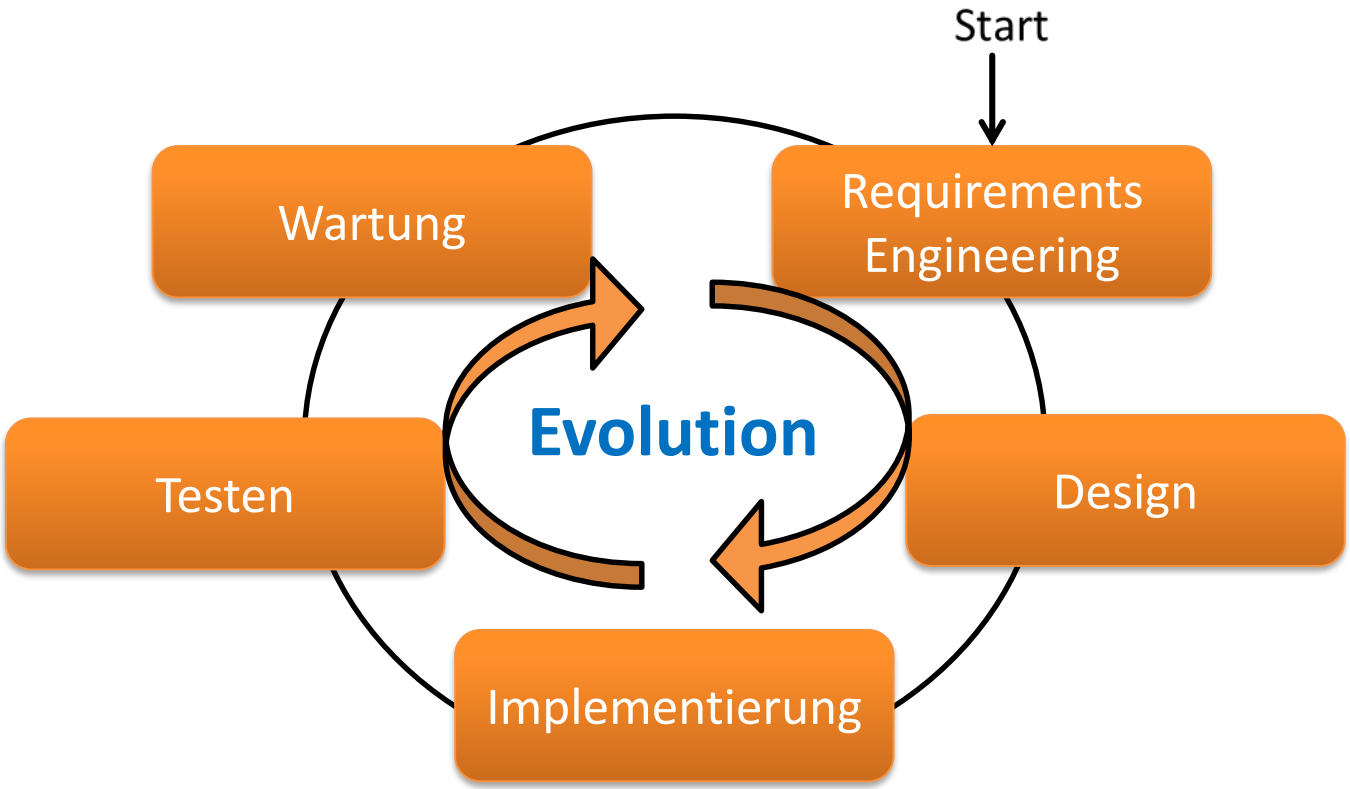
- Lange Methode ist Zeichen dafür, wenn man
  - Zu viele Dinge auf einmal machen möchte
  - Nicht genug über Abstraktionen und Grenzen nachgedacht hat

- Fowler's Heuristik:
  - Bei Kommentar neue Methode beginnen
  - Kommentar deutet an:
    - Nächster großer Schritt
    - Etwas nicht-offensichtliches, dessen Details von Klarheit der Methode ablenken
  - In beiden Fällen gute Stelle zum Teilen

Evolution



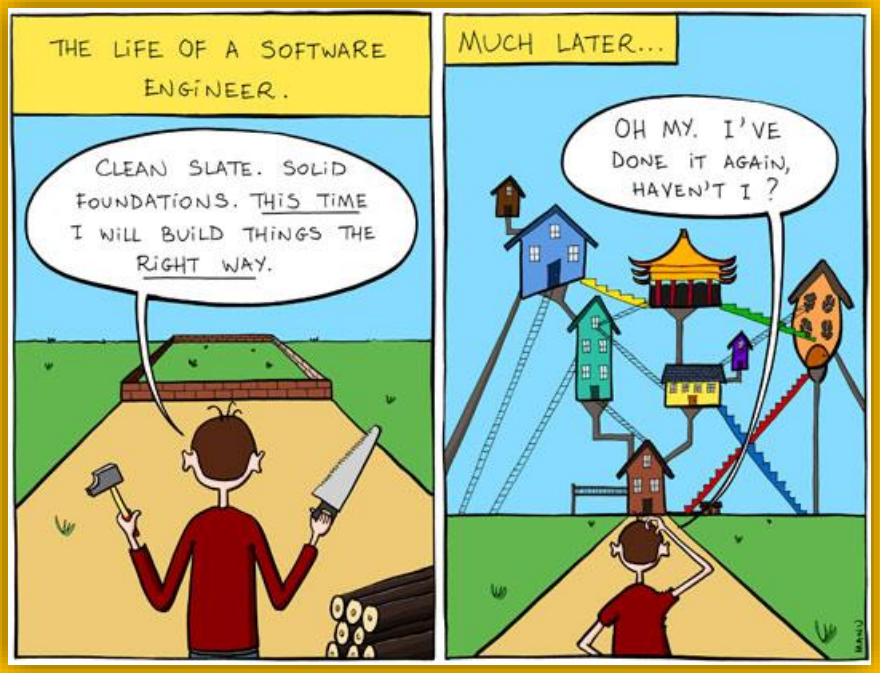
Einordnung


Start
Wartung
Requirements Engineering
Evolution
Testen
Design
Implementierung

Lernziele

- Bedeutung von Software Evolution verstehen
- Gesetze der Software Evolution kennen lernen
- Erlernen wichtiger Methoden zur Begegnung der Evolution
  - Reengineering Engineering
  - Refactoring
  - Software Repositories

Warum ist Evolution wichtig?



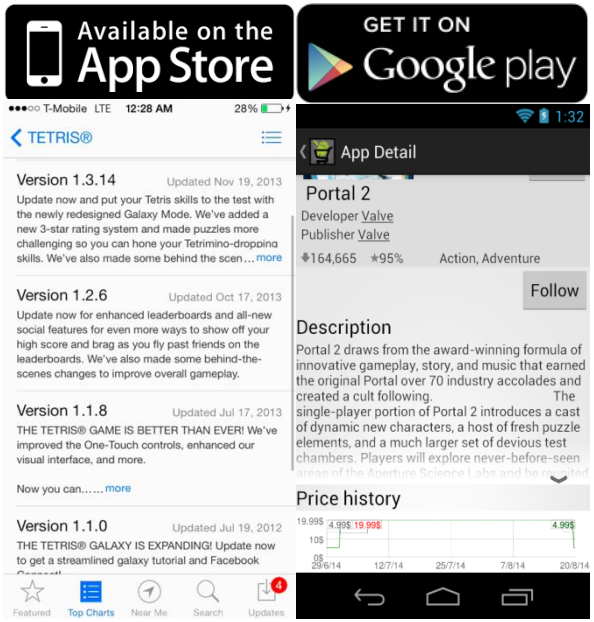
Was ist Software Evolution?

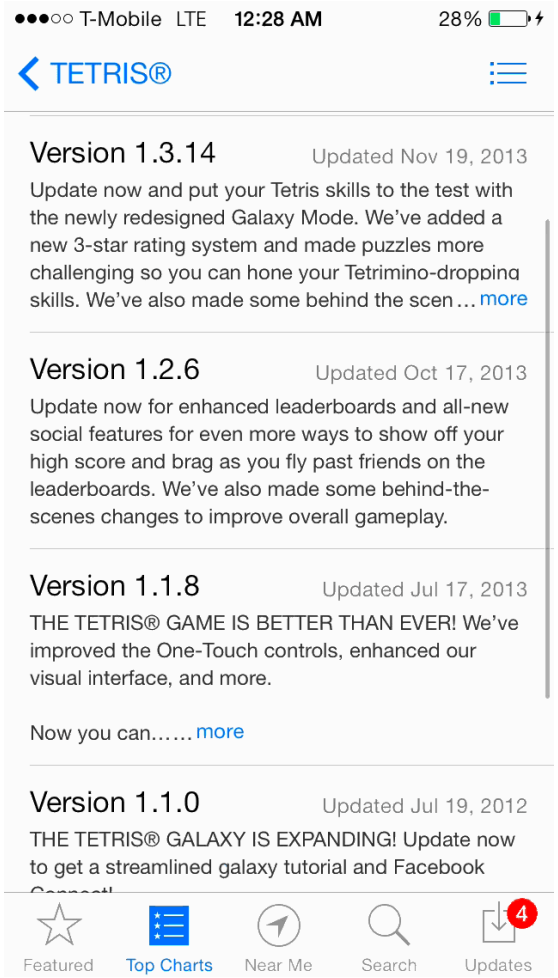
Software Evolution ist ein Prozess, bei dem ein erfolgreiches Software System kontinuierliche Änderungen zur Verbesserung, Funktionsanreicherung sowie Fehlerbehebung erfordert.

Aber was ist Software Evolution in der Praxis?

Evolution in nahezu jedem Software System

Mobile Apps





Evolution in nahezu jedem Software System

Mobile Apps

Games

Classic · The Burning Crusade · Wrath of the Lich King · Cataclysm · Mists of Pandaria · Warlords of Draenor

Pre-release 0.x	0.6 · 0.7 · 0.8 · 0.9 · 0.9.1 · 0.10 · 0.11 · 0.12
WoW Classic 1.x	**1.1.0** · 1.1.1 · 1.1.2 – **1.2.0** · 1.2.1 · 1.2.2 · 1.2.3 · 1.2.4 – **1.3.0** · 1.3.1 · 1.3.2 – **1.4.0** · 1.4.1 · 1.4.2 – **1.5.0** · 1.5.1 – **1.6.0** · 1.6.1 – **1.7.0** · 1.7.1 – **1.8.0** · 1.8.1 · 1.8.2 · 1.8.3 · 1.8.4 – **1.9.0** · 1.9.1 · 1.9.2 · 1.9.3 · 1.9.4 – **1.10.0** · 1.10.1 · 1.10.2 – **1.11.0** · 1.11.1 · 1.11.2 – **1.12.0** · 1.12.1 · 1.12.2
Burning Crusade 2.x	**2.0.1** · **2.0.3** · 2.0.4 · 2.0.5 · 2.0.6 · 2.0.7 · 2.0.8 · 2.0.9 · 2.0.10 · 2.0.11 · 2.0.12 – **2.1.0** · 2.1.1 · 2.1.2 · 2.1.3 · 2.1.4 – **2.2.0** · 2.2.2 · 2.2.3 – **2.3.0** · 2.3.2 · 2.3.3 – **2.4.0** · 2.4.1 · 2.4.2 · 2.4.3
Wrath of the Lich King 3.x	3.0.1 β · **3.0.2** · 3.0.3 · 3.0.5 · 3.0.8 · 3.0.8a · 3.0.9 – **3.1.0** · 3.1.1 · 3.1.1a · 3.1.2 · 3.1.3 – **3.2.0** · 3.2.0a · 3.2.2 · 3.2.2a – **3.3.0** · 3.3.0a · 3.3.2 · 3.3.3 · 3.3.3a · 3.3.5 · 3.3.5a
Cataclysm 4.x	**4.0.1** · 4.0.1a · 4.0.3 · *4.0.3a* · 4.0.6 · 4.0.6a – **4.1.0** · 4.1.0a · 4.1.11 – **4.2.0** · 4.2.0a · 4.2.2 – **4.3.0** · 4.3.0a · 4.3.2 · 4.3.3 · 4.3.4 – 4.4.0
Mists of Pandaria 5.x	5.0.1 β · 5.0.3 · **5.0.4** · 5.0.5 · 5.0.5b – **5.1.0** · 5.1.0A – **5.2.0** · 5.2.0a · 5.2.0b · 5.2.0c · 5.2.0d · 5.2.0e · 5.2.0f · 5.2.0g · 5.2.0h · 5.2.0i · 5.2.0j – **5.3.0** · 5.3.0hotfix1 · 5.3.0hotfix2 · 5.3.0hotfix43 · 5.3.0hotfix4 · 5.3.0a – **5.4.0** · 5.4.0hotfix1 · 5.4.0hotfix2 · 5.4.1 · 5.4.2 · 5.4.2hotfix1 · 5.4.7 · 5.4.7hotfix1 · 5.4.7hotfix2 · 5.4.7hotfix3 · 5.4.8 – 5.5.0
Warlords of Draenor 6.x	6.0.0 (Test-only) · 6.0.1 (Test-only: α &amp β) · 6.0.2 (β, PTR, & pre-patch) · 6.0.3 (β & pre-patch) · 6.0.3 hotfix1 (pre-patch) · 6.0.3 hotfix2 (pre-patch) · 6.0.3 hotfix3 (pre-patch & expansion) · 6.0.3 hotfix4 · 6.0.3 hotfix5 · 6.1.0 · 6.1.0 hotfix1 · **6.1.2**
Removed content	1.1.0 · 1.2.0 · 1.3.0 · 1.4.0 · 1.5.0 · 1.6.0 · 1.6.1 · 1.7.0 · 1.8.0 · 1.9.0 · 1.10.0 · 1.11.0 · 1.11.2 · 2.0.1 · 2.0.3 · 2.0.4 · 2.1.0 · 2.2.0 · 2.3.0 · 2.4.0 · 2.4.3 · 3.0.2 · 3.0.3 · 3.0.8 · 3.1.0 · 3.2.0 · 3.2.2 · 3.3.0 · 3.3.3 · 4.n.n · 4.0.1 · 4.0.3 · 4.0.3a · 4.1.0 · 4.2.0 · 4.3.0 · 5.n.n · β 5.0.1 · β 5.0.3 · 5.0.4 · 5.0.5 · 5.1.0 · 5.2.0 · 5.3.0 · 5.4.0
Hotfixes	2006 · 2007 · 2008 · 2009 · 2010 · 2011 · 2012 · 2013 · **2014**

Evolution in nahezu jedem Software System

Mobile Apps

Games

Software Systeme

1935: Z1

1972: C

1986: 32-Bit CPU

1995: Java

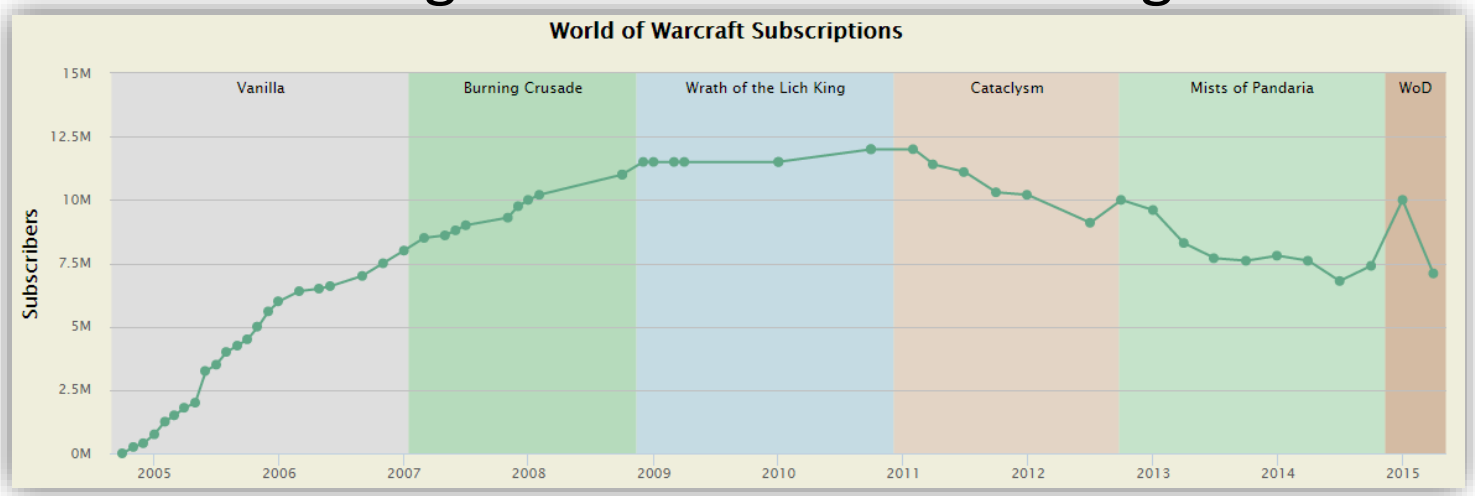
2007: iPhone

Gründe für Software Evolution

- Was sind Gründe der Software Evolution für folgende Software Systeme:
  - Mobile Apps,
  - Games,
  - Infrastruktur-Software (Datenbanken, Betriebssysteme, etc.),
  - Eingebettete Software (für Flugzeuge, Autos, etc.) ?

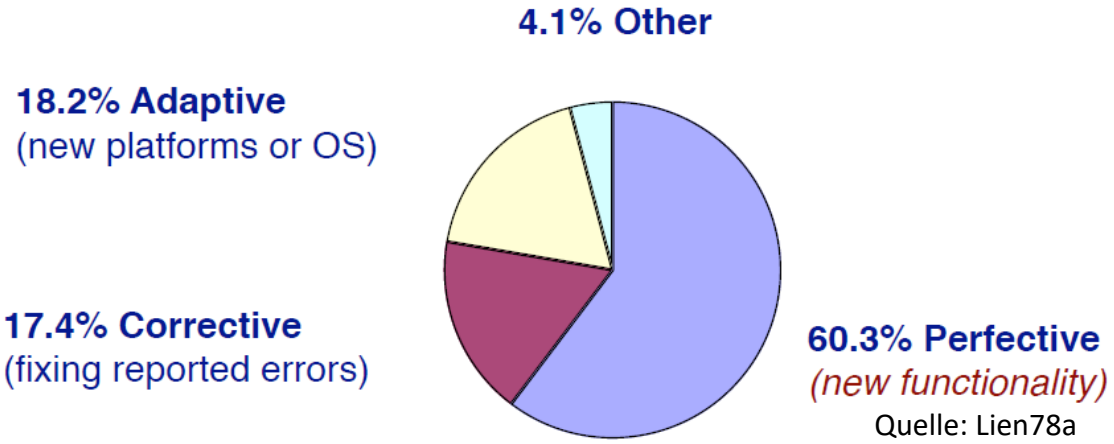
Gründe für Software Evolution

- Kundenbindung und ökonomischer Erfolg



  - Neue Anforderungen (von Kunden, Hardware, Software)
  - Fehlerbehebung / Verbesserung (Performanz, Usability)
- Verkürzte time-to-market (insb. im mobilen Sektor)
- Langlebiger Einsatz

Arten der Aufgaben



Quelle: Lien78a

- Überwiegender Teil der Evolutionskosten für neue Funktionalität (auch Verbesserung von Qualitätseigenschaften)
- Einer der Gründe für neuartige, *agile* Entwicklungsmethoden (nächste Vorlesung)

Gesetze der Evolution: Lehman‘s Laws

- Acht Gesetze:
  - Anhaltender Wandel
  - Zunehmende Komplexität
  - Selbstregulation
  - Erhaltung der organisatorischen Stabilität
  - Erhaltung der Vertrautheit
  - Anhaltendes Wachstum
  - Sinkende Qualität
  - Feedback-System

Ein System, welches benutzt wird, unterliegt kontinuierlichen Veränderungen. Änderungen erhöhen Komplexität und verringern Struktur des Systems.

Was sind Probleme der Evolution und wie können wir sie lösen?

- Reegineering
- Refactoring

Problem: Software Ageing

- *Veraltete* oder *keine* Dokumentation
- *Ursprüngliche* Entwickler *nicht* mehr *verfügbar*
- *Mangelhaftes Verständnis* über das System (Tests fehlen oder sind unvollständig, fehlerhaftes Mapping zwischen Anforderungen und Code)

Reengineering

Probleme: Steigende Entwicklungszeit

- *Ständige* Fehlerbehebungen
- Entwicklung neuer Features *dauert lange*
- Änderungen am Code sind *aufwändig*

- Code Smells (z.B. duplizierter Code, kaum Modularität)
- Antipatterns (z.B. duplizierte Funktionalität)

Refactoring

Lösung: Design for Change!

- Berücksichtige mögliche Erweiterungen im *Design* (siehe Design Patterns) und *Architektur*

- Baue nicht die eierlegende Wollmilchsau, sondern eine *Familie von Software Systemen* (siehe Produktlinienvorlesung)

- Falls vorbeugende Maßnahmen nicht ausreichen:
  - Reengineering
  - Refactoring

Software Reengineering

"*Reengineering* ... is the *examination* and *alteration* of a subject system to *reconstitute* it in a new form and the subsequent *implementation* of the new form."

Chikofsky and Cross

[R.S. Arnold, Software Reengineering, IEEE CS Press, 1993]

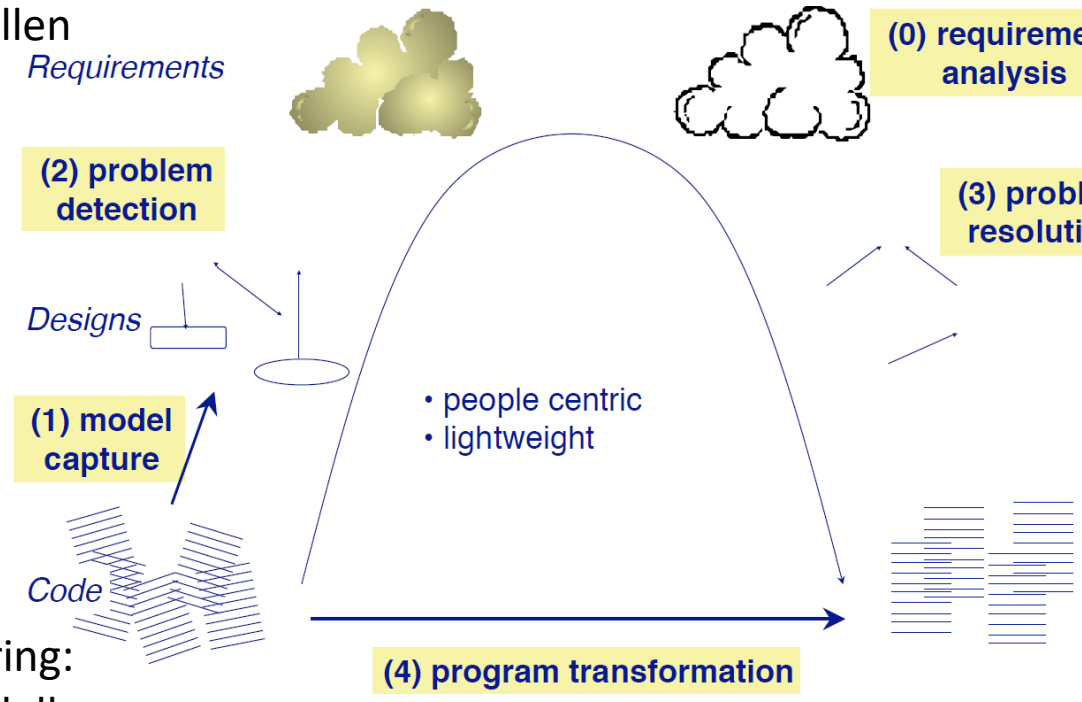
Rekonstruktion (Teile) eines Systems, um Wartbarkeit und Verständlichkeit zu verbessern.

Ziele des Reengineerings

- *Untangling (dt.: entflechten / entwirren)*
  - Teile ein monolithisches System in Module auf, die einzeln besser gewartet werden können
  - Erhöht die Verständlichkeit des Codes
- *Performance*
  - Zuerst tue es, dann tue es richtig und zuletzt tue es schnell! — Erfahrung hat gezeigt, dass dies die richtige Reihenfolge ist!
- *Portierung*
  - Die Architektur muss die plattformabhängigen Module unterscheiden
- *Design Extraktion*
  - Verbessert Wartbarkeit, Portabilität, etc.
- Ausnutzung neuer Technologien
  - Neue Sprachfeatures, Standards, Bibliotheken, etc.

Reengineering Lebenszyklus

Identifiziere *Design Probleme* in den abstrakten Modellen



Welche Teile der Anforderungen haben sich *geändert*?

Schlage *alternatives Design* vor, welches die identifizierten Probleme löst

Reverse Engineering: Von Code zu Modellen (hinzu *abstrakter Beschreibung*)

Mache die *notwendigen Änderungen* am Code, so dass das neue Design verwendet wird, jedoch alle *benötigte Funktionalität erhalten bleibt*. Erfordert testen!

Ziele des Reverse Engineerings

- Ermögliche *Wiederverwendung*
  - Identifiziere wiederverwendbare Kandidaten (Komponenten, etc.)
- Generiere *alternative Sichten*
  - Unterschiedliche Wege das selbe System zu betrachten
- Synthetisiere *höhere Abstraktionen*
  - Identifiziere mögliche Abstraktionen in der Software

- Meistere *Komplexität*
  - Techniken, um große, komplexe Systeme zu verstehen
- Finde *verlorene Informationen* wieder
  - Extrahiere, welche Änderungen gemacht wurden und warum
- Detektiere *Nebenläufigkeiten*
  - Verbessere Verständnis von Auswirkungen von Änderungen

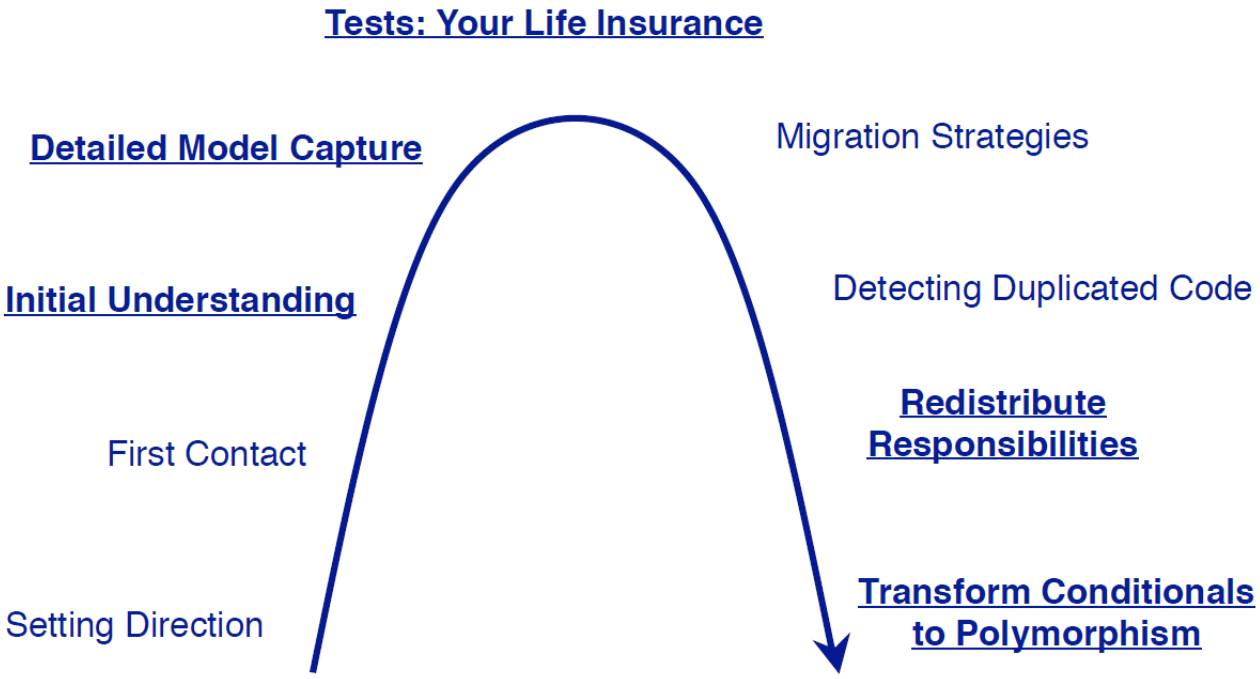
— Chikofsky and Cross [in Arnold, 1993]

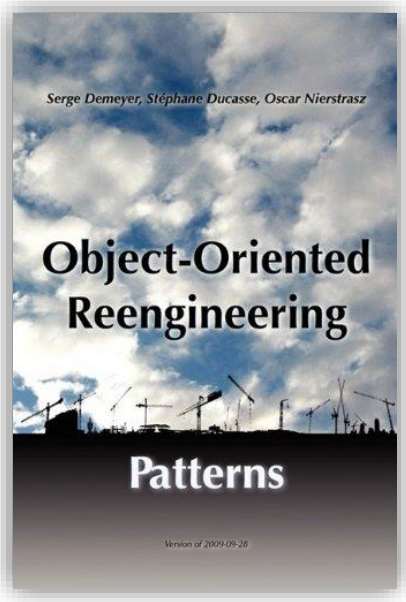
Reverse Engineering – Techniken

- Re-Dokumentation
  - Diagrammgeneratoren (aus Code)
  - Generatoren für Querverweise (über Module hinweg)

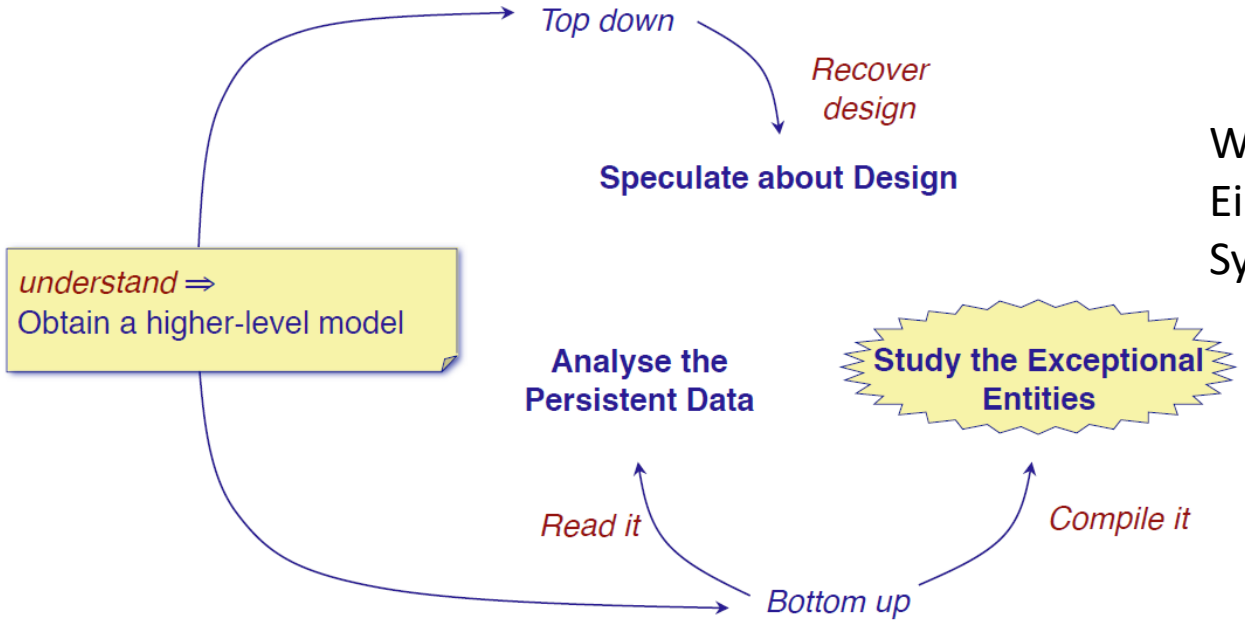
- Design Recovery
  - Software Metriken
  - Visualisierungstools
  - Statische Analysierer
  - Dynamische Analysierer

Reengineering Patterns





Initiales Verständnis



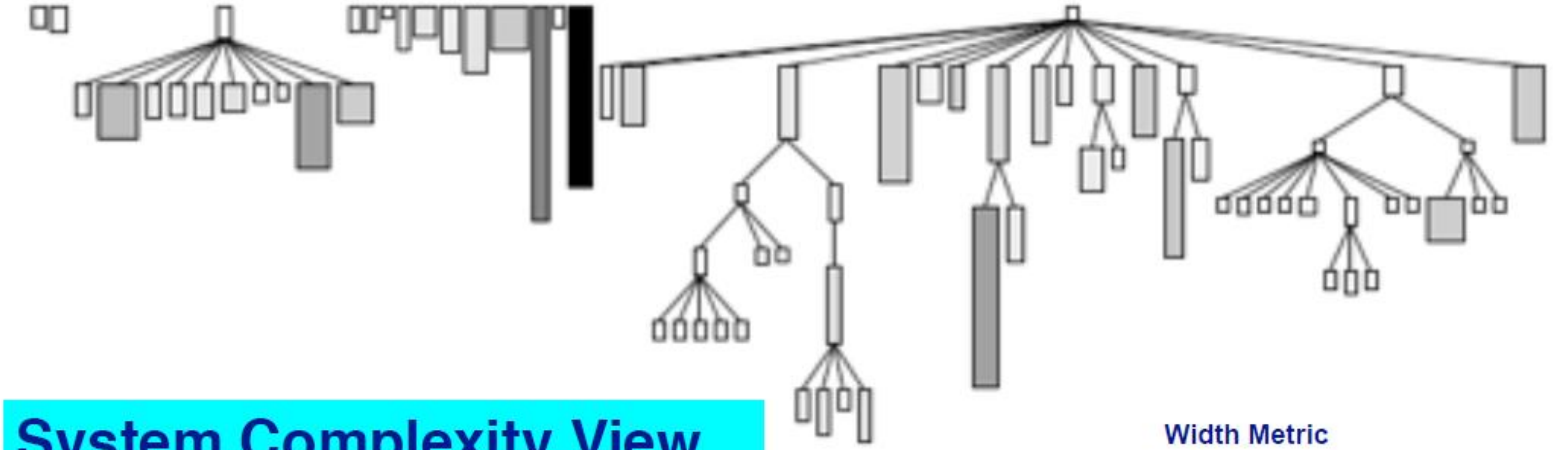
Wie können wir schnell Einsichten in ein komplexes System bekommen?

Lösung:

Messe Software und *studiere Anomalien*!

- Nutze simple Metriken
- Visualisiere Metriken
- Begutachte Code, um Einsichten über Anomalien zu erhalten

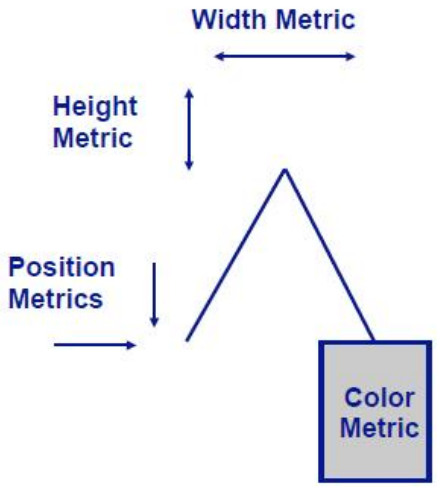
Wie helfen Metriken dabei?



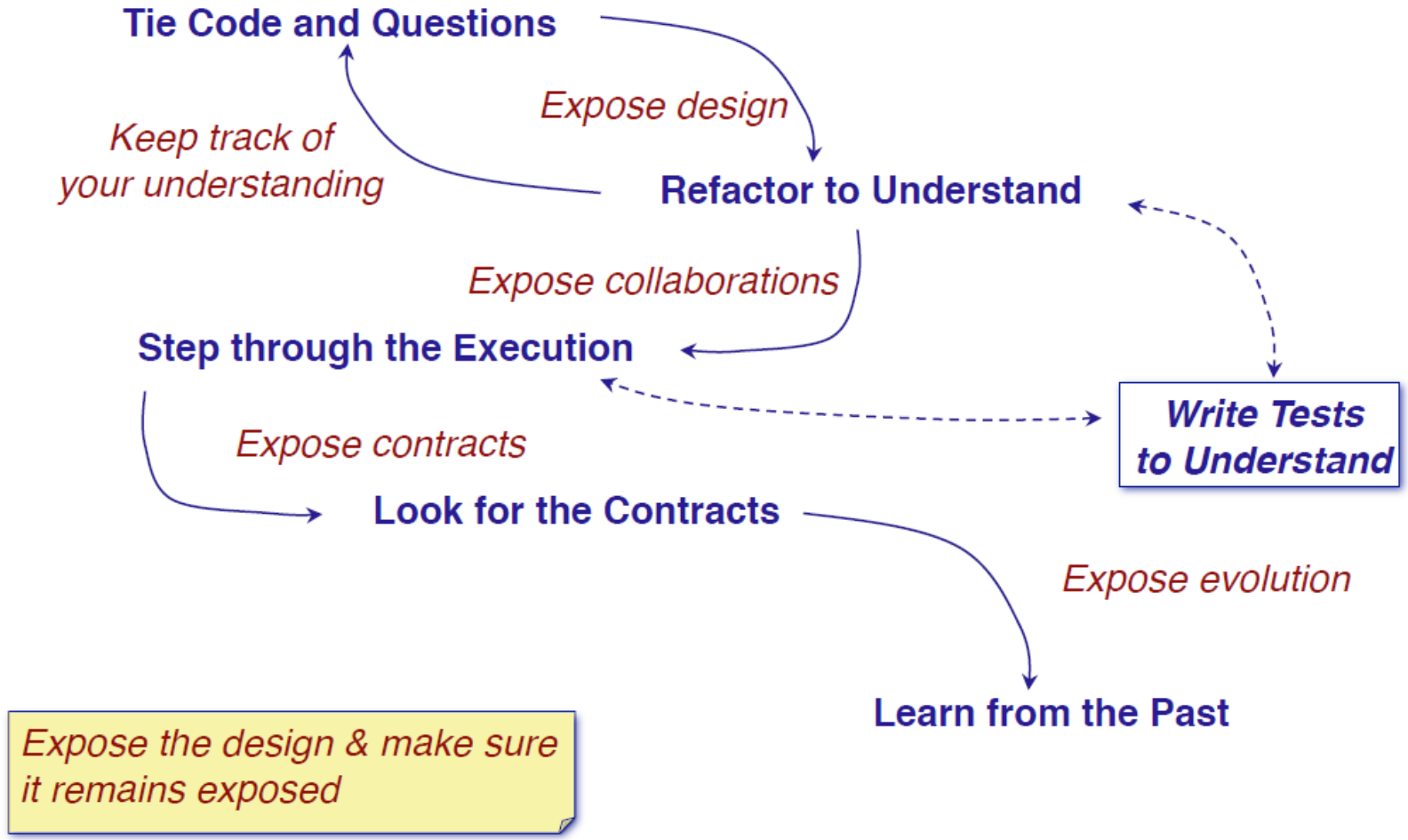
System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

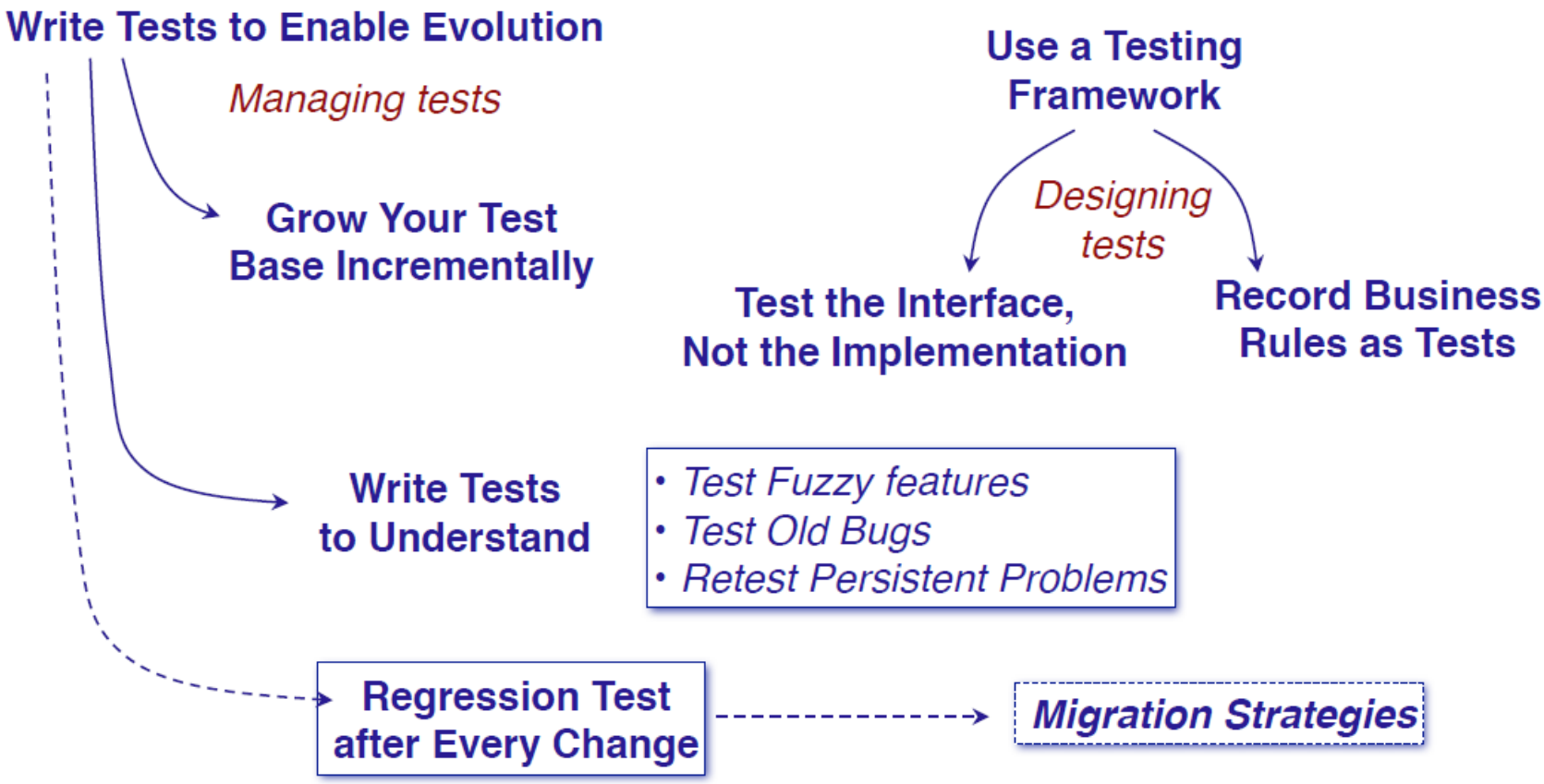
Width = Number of Attributes
Height = Number of Methods
Color = Number of Lines of Code



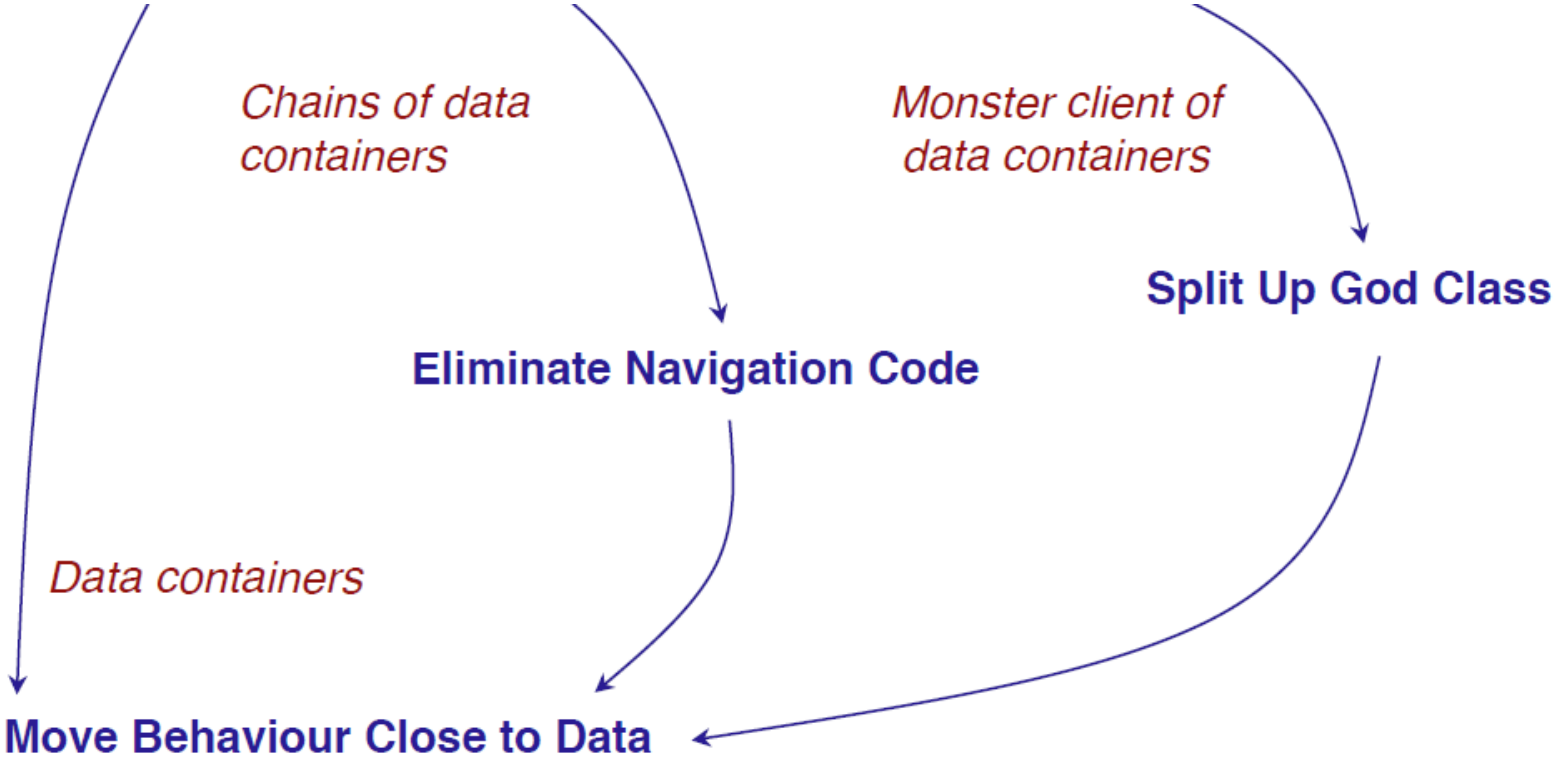
Auf dem Weg zu einem detaillierten Modell



Schreibe Tests, um Systemverhalten und alte Designentscheidungen zu verstehen

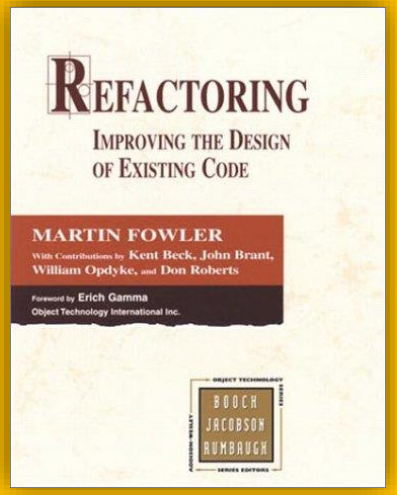


Verteile Verantwortlichkeiten neu



Refactoring

"The process of *changing a software system* in such a way that it *does not alter the external behaviour* of the code, yet *improves its internal structure*."



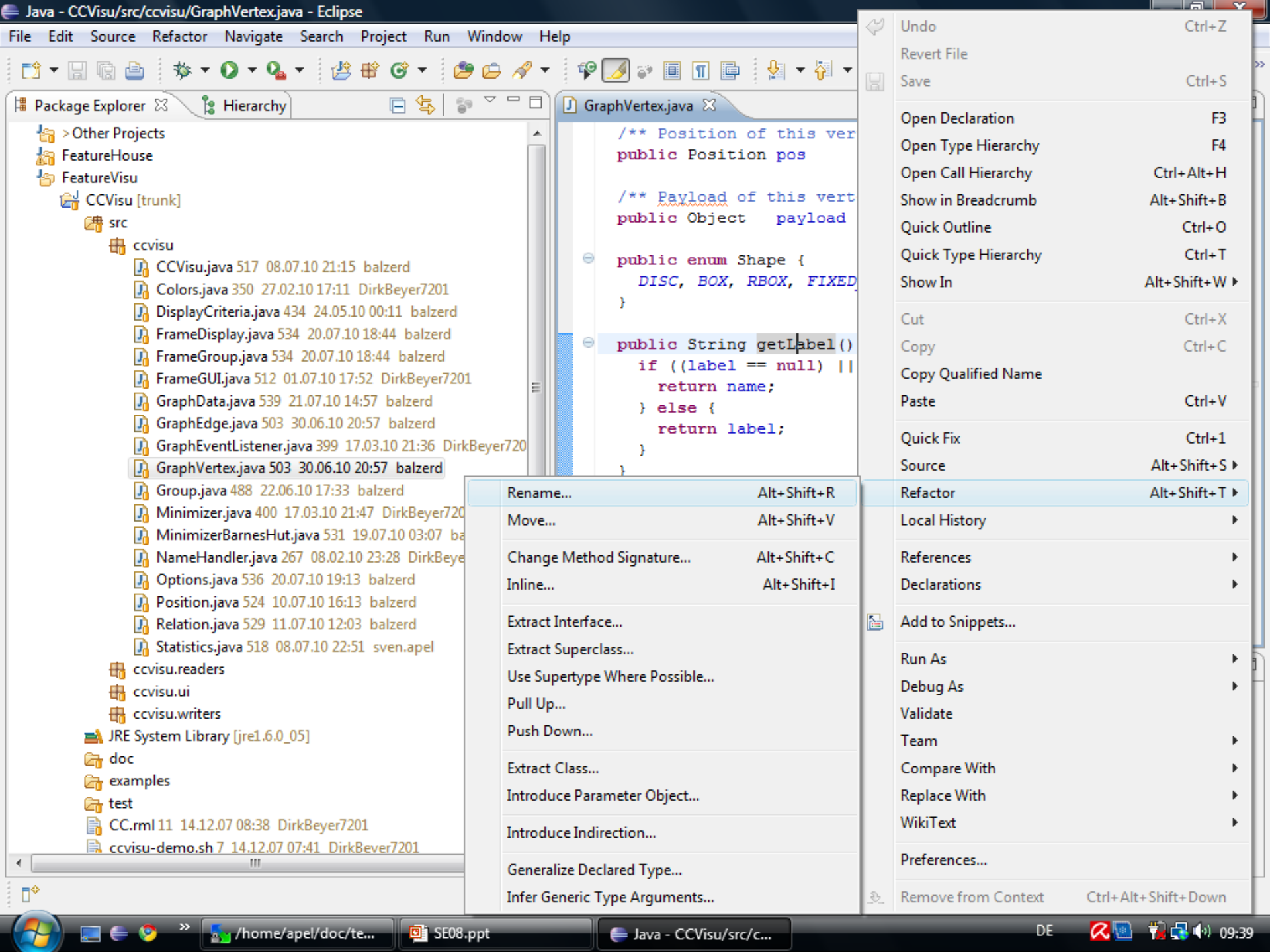
Fowler, et al., Refactoring, 1999

Warum Refactoring?

- „Grow, don‘t build software“ (Fred Brooks)
- Realität:
  - Schwer die richtigen Designentscheidungen von Anfang an zu treffen
  - Schwer die Problemdomäne und die Nutzeranforderungen zu verstehen
  - Schwer Vorherzusehen, wie das System in 5 Jahren wächst
- Refactoring hilft
  - Code zu ändern, ohne das Verhalten zu ändern
  - Situationen zu schaffen, wo Evolution sicher ablaufen kann
  - Code zu verstehen

The Rule of Three (XP)

- Das erste Mal, wenn du eine Aufgabe implementierst, *tue es einfach*.
- Das zweite mal, wenn du die gleiche Aufgabe implementierst, *zucke zusammen* und mache es wieder.
- Das dritte Mal, wenn du die gleiche Aufgabe implementierst, ist es Zeit für *Refactoring*!
  - Finde richtige Mischung aus Abstraktion von Programmkonstrukten (erhöht Flexibilität) und praktischem Nutzen


Java - CCVisu/src/ccvisu/GraphVertex.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy
Other Projects
FeatureHouse
FeatureVisu
CCVisu [trunk]
src
ccvisu
CCVisu.java 517 08.07.10 21:15 balzerd
Colors.java 350 27.02.10 17:11 DirkBeyer7201
DisplayCriteria.java 434 24.05.10 00:11 balzerd
FrameDisplay.java 534 20.07.10 18:44 balzerd
FrameGroup.java 534 20.07.10 18:44 balzerd
FrameGUI.java 512 01.07.10 17:52 DirkBeyer7201
GraphData.java 539 21.07.10 14:57 balzerd
GraphEdge.java 503 30.06.10 20:57 balzerd
GraphEventListener.java 399 17.03.10 21:36 DirkBeyer720
GraphVertex.java 503 30.06.10 20:57 balzerd
Group.java 488 22.06.10 17:33 balzerd
Minimizer.java 400 17.03.10 21:47 DirkBeyer720
MinimizerBarnesHut.java 531 19.07.10 03:07 ba
NameHandler.java 267 08.02.10 23:28 DirkBeye
Options.java 536 20.07.10 19:13 balzerd
Position.java 524 10.07.10 16:13 balzerd
Relation.java 529 11.07.10 12:03 balzerd
Statistics.java 518 08.07.10 22:51 sven.apel
ccvisu.readers
ccvisu.ui
ccvisu.writers
JRE System Library [jre1.6.0_05]
doc
examples
test
CC.rml 11 14.12.07 08:38 DirkBeyer7201
ccvisu-demo.sh 7 14.12.07 07:41 DirkBeyer7201
GraphVertex.java
/** Position of this ver
public Position pos
/** Payload of this vert
public Object payload
public enum Shape {
DISC, BOX, RBOX, FIXED
}
public String getLabel()
if ((label == null) ||
return name;
} else {
return label;
}
}
Rename... Alt+Shift+R
Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Inline... Alt+Shift+I
Extract Interface...
Extract Superclass...
Use Supertype Where Possible...
Pull Up...
Push Down...
Extract Class...
Introduce Parameter Object...
Introduce Indirection...
Generalize Declared Type...
Infer Generic Type Arguments...
Undo Ctrl+Z
Revert File
Save Ctrl+S
Open Declaration F3
Open Type Hierarchy F4
Open Call Hierarchy Ctrl+Alt+H
Show in Breadcrumb Alt+Shift+B
Quick Outline Ctrl+O
Quick Type Hierarchy Ctrl+T
Show In Alt+Shift+W
Cut Ctrl+X
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Quick Fix Ctrl+1
Source Alt+Shift+S
Refactor Alt+Shift+T
Local History
References
Declarations
Add to Snippets...
Run As
Debug As
Validate
Team
Compare With
Replace With
WikiText
Preferences...
Remove from Context Ctrl+Alt+Shift+Down
/home/apel/doc/te...
SE08.ppt
Java - CCVisu/src/c...
DE
09:39

Code Smells relevant für Refactoring

- Fowlers Buch hat einen Katalog von:
  - 22 “bad smells” (also Antipatterns)
    - Duplicated code
    - Long method
    - Large class
    - Long parameter list
    - Divergent change
    - Shotgun surgery
    - …
  - 72 “refactorings” (also was zu tun ist, wenn Antipattern gefunden wurde)

Typische Refactorings

Class	Method	Attribute
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Was Sie mitgenommen haben sollten:

- Würden Sie Softwaremetriken benutzen, um Code-Qualität zu bewerten? Warum?
- Erläutern Sie, warum unterschiedliche Stakeholder zu Problemen bei der Definition und Einhaltung von Softwarequalität führen können.
- Nennen Sie X Software Qualitätsmerkmale. Wie würden Sie die Einhaltung dieser Attribute sicherstellen? Warum?
- Sind Qualitätsstandards sinnvoll? Warum?
- Nennen/erklären Sie die verschiedenen Wartungsaktivitäten.
- Erklären Sie X typische Code Smells/Antipatterns und beschreiben Sie deren Behebung.
- Warum ist es sinnvoll, sich mit Code Smells und Antipatterns zu befassen?

Was Sie mitgenommen haben sollten:

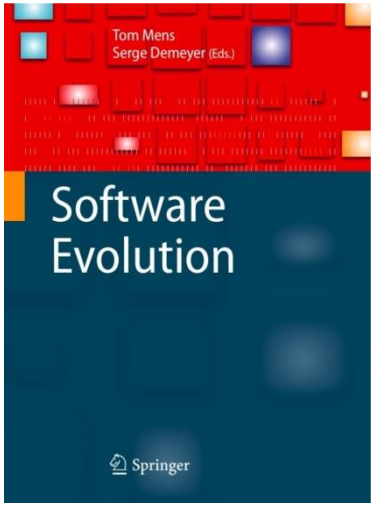
- Was sind die Gesetze von Lehman und wiese gelten sie noch heute?
- Warum werden Software Systeme immer komplexer über die Zeit?

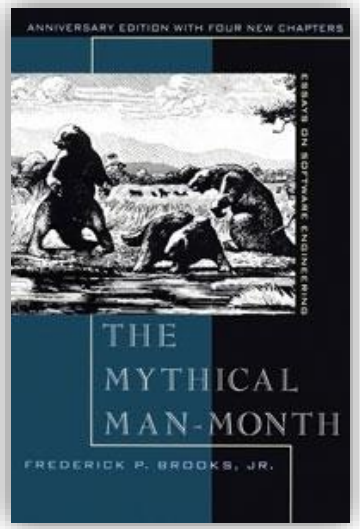
- Wann sollte man eher das ältere System behalten als ein neues zu schreiben?
- Was bedeutet Reengineering und welche Phasen hat es?

- Welche Design Pattern unterstützen die Software Evolution?

Literatur

- *Designing Object-Oriented Software*, R. Wirfs-Brock, B. Wilkerson, L. Wiener, Prentice Hall, 1990.
- *Metrics and laws of software evolution—the nineties view*, Meir Lehman, Metrics, IEEE, 1997.
- Martin Fowler. Refactoring: Improving the Design of Existing Code
- Making Software. What Really Works, and Why We Believe It. 2010.





Folien basieren auf dem Script von Oscar Nierstrasz